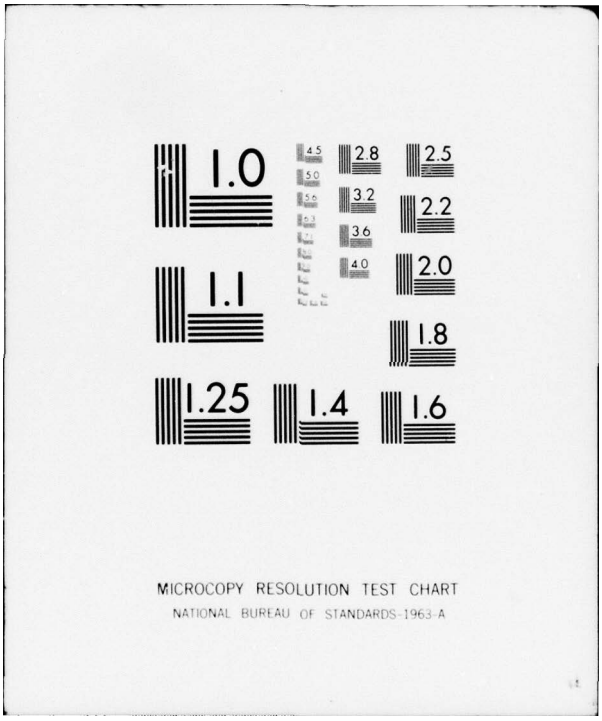
AD-A040 553 MASSACHUSETTS INST OF TECH LEXINGTON LINCOLN LAB F/G 9/2
A GENERAL-PURPOSE CROSS-ASSEMBLER FOR PRODUCING ABSOLUTE BINARY--ETC(U)
APR 77 P R KRETZ F19628-76-C-0002
UNCLASSIFIED TN-1977-20 ESD-TR-77-72 NL

| OF |
AD
A040553

END

DATE
FILMED
7-77


1.0
1.1
1.25
4.5
5.0
5.6
6.3
7.1
8.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
1.4
1.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

ESD-TR-77-78

Lee 1473 122

Technical Note

1977-20

P. R. Kretz

A General-Purpose Cross-Assembler for Producing Absolute Binary Object Code

6 April 1977

Prepared for the Department of the Air Force
under Electronic Systems Division Contract F19628-76-C-0002 by

Lincoln Laboratory

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LEXINGTON, MASSACHUSETTS

The work reported in this document was performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology, with the support of the Department of the Air Force under Contract F19628-76-C-0002.

This technical report has been reviewed and is approved for publication.

FOR THE COMMANDER

Raymond L. Loiselle

Raymond L. Loiselle, Lt. Col., USAF
Chief, ESD Lincoln Laboratory Project Office

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LINCOLN LABORATORY

A GENERAL-PURPOSE CROSS-ASSEMBLER FOR PRODUCING ABSOLUTE BINARY OBJECT CODE

P. R. KRETZ

Group 46

TECHNICAL NOTE 1977-20

6 APRIL 1977

LEXINGTON MASSACHUSETTS

ABSTRACT

A general-purpose cross-assembler is described. The cross-assembler, written in PL/I, has been implemented on an IBM 370/168 using the time-sharing Conversational Monitor System (CMS). Absolute binary object code will be produced. Although the cross-assembler has been designed with the intention of assembling code for various microprogrammable machines, even code for conventional minicomputers has been assembled.

Use of the cross-assembler is discussed assuming a CMS environment. Included are the disk-resident files to facilitate an assembly. Various pseudo-ops, or assembler control statements, are used to describe the machine for which an assembly is done. An example of using the cross-assembler for a parallel microprogrammable digital signal processor (PMP-I) is discussed.

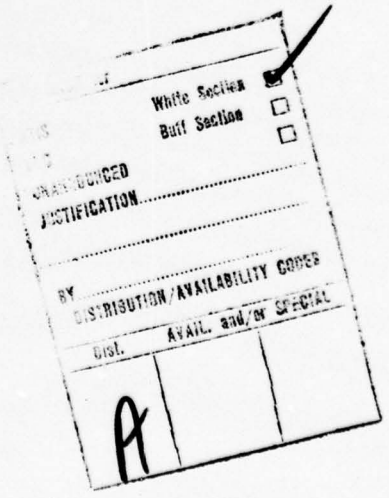


Table of Contents

Table of Contents (Continued)

INTRODUCTION

A general-purpose absolute cross-assembler has been written in PL/I to run on an IBM 370/168, accept metalanguage commands to define a machine, and then cross-assemble code for that user-defined machine. The cross-assembler has been used under IBM's time-sharing Conversational Monitor System (CMS), and this report will discuss the cross-assembler in that environment.

To begin, let us examine the evolutional history of this cross-assembler. Early in 1975, work was begun at M.I.T. Lincoln Laboratory on a Parallel Microprogrammable Processor (known as the PMP or more specifically, PMP-I).[1] This machine was designed to be a high-speed, programmable, signal processor. In parallel with the construction of the hardware, a set of microinstructions for the PMP was developed.[2] This led to the generation of a cross-assembler written in PL/I to run on the IBM 370/168 to assemble mnemonics and produce object for the PMP[3,4].

As time progressed, a new version of this processor, namely PMP-II, was proposed. This coupled with the increasing use of programmable microprocessors in other applications lead to the creation of a general-purpose cross-assembler which accepts as input metalanguage commands which contain such information as word length, memory size, bit meanings, etc. This, then is the version of the cross-assembler to be discussed.

The remainder of this memo is divided essentially into two major sections. The first of these sections attempts to look at the cross-assembler in the general sense, and attempts to explain the metalanguage commands necessary to assemble code for a particular machine. It begins by

explaining the input language rules anticipated by the cross-assembler. This section may seem unusual to those familiar only with minicomputer code, since the cross-assembler allows several microinstructions to be coded on the same line and assembles these into the same program memory word.

This is followed by a discussion of the files which should be created to cross-assemble code for a machine. These files typically contain all the necessary metalanguage statements, thus the programmer is not required to repeat these at the beginning of all programs. The available pseudo-ops are then discussed in depth, explaining when, where, and why they should appear. Section I is completed with a discussion of the format of the object binary file output by the cross-assembler. This section should be extremely helpful for the user who must create a program to convert cross-assembler output to some medium which is acceptable input to the machine for which cross-assembly occurred.

The second section of this report is dedicated to an example of the cross-assembler in action. Naturally, due to the parentage of the cross-assembler (and familiarities of the author) the sample machine chosen for this example was PMP-I. It is hoped that this discussion, along with the control files for PMP-I appearing as appendices, will give the user enough information to extend the cross-assembler for the particular machine desired. This section should also serve as a user's guide for the PMP-I programmer.

At this point it should be stressed that this cross-assembler has been designed to produce absolute binary output. This implies that for a machine which does program counter relative addressing the burden of coding program counter relative constants is left to the user. One must subtract the value

of the current program counter in order to obtain a program counter relative constant (e.g., if LABEL is to be expressed relative to the program counter one would need to code LABEL-*).

Although it was the intent of this cross-assembler to assemble microcode, code for conventional minicomputers may also be assembled. In the latter case the user would define all instructions belonging to the same non-zero exclusivity set, thus indicating to the cross-assembler that only one instruction per program memory word will be allowed.

As a final note, perhaps some mention of the architecture of the cross-assembler itself is in order. The cross-assembler performs only one-pass, but a provision for forward referencing of variables (i.e., using a label before its definition) has been built in. This implies the cross-assembler must be able to access program memory locations which have been previously assembled, in order to resolve any forward references to a label at the time of definition. The solution to this has been to maintain a core-resident array of all possible program memory locations. Since this may require considerable working space for machines with large program memories, the user has control to specify via the .DARCH pseudo-op which portion of memory must be generated, and therefore which portion of memory is available for use.

I. Characteristics of the General Cross-Assembler

One of the features of the generalized version of the cross-assembler is that all variables dependent on the machine for which assembly is to be done must be defined via metalanguage commands at each assembly. In order to

minimize the burden placed on the programmer a method has been devised whereby the required metalanguage definitions may be prestored as a separate disk file, and input so as to seem transparent to the user.

This section is devoted to in-depth details of what a user must do in order to create an environment to facilitate assemblies for a given machine. Special attention is called to subsection C, which gives in-depth definitions for all pseudo-ops available, including the metalanguage commands available for defining a machine as well as pseudo-ops necessary for an assembly.

The first step to be accomplished is to divide the set of available instruction mnemonics into canonic classes (i.e., groups of instructions having similar operands). The canonic classes may then be defined by stating minimum and maximum number of allowable operands, indicating any required operands, and enumerating elements of the set of available operands. A set of immediate constant (IC) classes also needs to be defined. An IC class has a lower and upper limit, and an associated beginning and ending bit number in a program memory word. Once both the canonic instruction classes and the IC classes have been conceptualized, the user is ready to begin creating the appropriate disk files.

A. Assembler Input Language Specifications

A "line" of code to be assembled should be a character string consisting of 80 characters, of which only the first 72 will be examined, allowing for the use of sequence numbers.

Since the cross-assembler was designed primarily for assembling instructions for machines which may be microcoded, coding more than one

instruction per program memory word is allowable. Multiple instructions to be assembled into the same program memory word may be coded on the same line and separated with a semicolon (;). An alternate method to code multiple instructions for the same memory location is to code them on consecutive lines, with the first non-blank character in each line after the first being a comma (,). Note that using a comma as the first non-blank character does not indicate a continuation of the previous instruction, but rather a continuation of the program memory location, and therefore all lines should end at the end of an instruction.

A line will be scanned and all characters concatenated to form what is called a token, until a delimiting character is found. Delimiting characters are members of the set {blank ( ), semicolon (;), comma (,), colon (:), slash (/), equals (=)}. Tokens, then, are either labels, opcodes, or operands. Certain of the delimiters have special meaning. A slash always indicates that the remainder of the current line is a comment, a semicolon always indicates the end of an instruction. A colon appearing after the first token of a line indicates that the first token was actually a label, and a comma appearing as the first non-blank character indicates a continuation of the contents of the memory location being assembled on the previous line. Other than these few simple rules, the delimiting characters may be used interchangeably. Naturally, for cosmetic reasons a standard should be chosen for a machine and adhered to. No escape characters are provided, since no text string assembling is implemented.

When coding an immediate constant, the plus (+) and minus (-) signs have the normal unary and/or binary meanings. An immediate constant may contain any number of plus or minus signs, but no parenthesized expressions are accepted. For example, LABEL1+3, and LABEL1-LABEL2+3-LABEL3 are both acceptable immediate constants.

The current value of the address counter (as accumulated by the cross-assembler) is specified by an asterisk (*). Thus, immediate constants of the form program counter ± expression may be coded (e.g. *+2, *-3, *+LAB1, etc.). It should be noted that this merely produces an absolute immediate constant. If the user wishes to generate a program counter relative constant (i.e., the effective address is calculated at execution time by adding the program counter to the immediate constant field stored in the instruction) then at assembly time one should subtract the value of the program counter (e.g., LABEL-*, or LABEL+3-*).

Numbers may be input in any of 4 bases; binary, octal, decimal, or hexadecimal. The default input base is initially decimal, but may be changed via the .DIBASE pseudo-op. The default input base is merely the base used to evaluate a number which has no base indicator. At any time the user may code a number in any of these 4 bases by preceding the number with one of the base indicators (.B, .Q, .D, .H, or '). .B indicates a binary number, .Q an octal number, .D a decimal number, .H a hexadecimal number, and the apostrophe (') indicates an octal number. Thus, valid numbers might be: .B1101, .Q75, .D290, .H3AB, '72, or .HA2. Note that all numbers must either begin with a digit or one of the base indicators. Thus, if the default input base is

hexadecimal and the user wishes to code the constant A2, the constant 0A2 must be coded. However, .HA2 will also always be acceptable.

B. Using the General Cross-Assembler

Following is a discussion of the steps necessary to use the cross-assembler. This includes a description of the disk files which may be generated in order to ease the burden placed on the user. Creating an appropriate set of disk files frees the user from requiring machine definition pseudo-ops in all source programs.

B.1 Creating an EXEC File

Clearly the user will desire a specially tailored EXEC file to contain the necessary CP/CMS commands. The first four such commands should be:

```
&CONTROL ERROR
CP LINK PLIOPT 191 199 RR
ACCESS 199 Z/Z
GLOBAL TXTLIB SYSLIB PLILIB FORTLIB GRLL
```

The first of the above four statements is an EXEC control statement which specifies that the remaining CP/CMS commands should be typed at the terminal only if they result in a non-zero return code. The subsequent three commands access the disk containing the PL/I libraries necessary to run the cross-assembler.

A series of seven FILEDEF commands is then necessary. These may appear in any order. The following section attempts to describe the required files, each file discussed is referenced by its ddname.

1. ARCFILE - This file normally contains the required metalanguage pseudo-ops to define a particular machine (such as the .DARCH, .DCLASS's, .DOP's, etc). It is read before attempting to assemble the user's program. The only difference between this file and the source program to be assembled is that the input statements of ARCFILE are not written to the listing. As an example, if we have a file whose filename is PMP and whose filetype is ARCH, we would code:

```
FI ARCFILE DISK PMP ARCH (LRECL 80 BLOCK 800 RECFM FB
```

2. SFILE - This disk file is the source file to be assembled. It has been found convenient to let file-name, filetype, and filemode be respectively parameters 1, 2, and 3 of the EXEC. Thus, one would code:

```
FI SFILE DISK &1 &2 &3 (LRECL 80 BLOCK 800 RECFM FB
```

3. ADRFILE - This is an output disk file which will contain the address indices for the object binary file. The name GENADR DATA has typically been used, although if something like &1 DATA were used the object binary could be referenced even after another file was cross-assembled. Currently used is:

```
FI ADRFILE DISK GENADR DATA (BLOCK 80 RECFM VB
```

4. WFILE - This is another output file, the listing file. In order to write this file on disk with the same filename as that of the source file and with filetype LISTING one would code:

```
FI WFILE DISK &1 LISTING (LRECL 133 BLOCK 133 RECFM F
```

5. BINFILE - This file will contain the object binary as produced by the cross-assembler. In order to create this file as a disk file with the same filename as the source and with filetype OBJECT one would code:

```
FI BINFILE DISK &1 OBJECT (LRECL 80 BLOCK 800
```

6. TFILE - This output file will consist of any error messages and lines producing these errors, and a statement saying how many errors were produced during assembly. The file was intended to be sent to the terminal, which is done by:

```
FI TFILE TERMINAL (LRECL 132
```

7. INSFILE - This file should contain the instruction table as produced by the cross-assembler by the .CREATI pseudo-op. The pseudo-op .DEFLTI makes INSFILE an input file, and .CREATI makes INSFILE an output file. If we wished this file to have a filename of GENINS and filetype DATA we would code:

```
FI INSFILE DISK GENINS DATA (LRECL 80 BLOCK 800 RECFM FB
```

After all of the above FILEDEF commands the LOAD command completes the EXEC file. This command is given by:

LOAD GENASM GENHASH GENBILD (NOMAP NODUP START

B.2 Generating an Instruction Table

Generating an instruction table which is written to disk can save the user a considerable amount of time when assembling. The table consists of all defined instruction mnemonics, and control information for the cross-assembler.

In order to create this table the user must assemble a file which consists solely of three pseudo-ops: .DARCH, .DEFI, and .CREATI. Naturally since all assemblies must begin with a .DARCH this pseudo-op should be the first statement to be read. This should be followed by a .DEFI for each mnemonic instruction to be defined. (See Section C for use of the .DEFI pseudo-op.) Since the only two pseudo-ops automatically built into the instruction table are .DARCH and .DEFI the remaining pseudo-ops must also be defined with a .DEFI pseudo-op. (For the exact definitions necessary for these see Appendix H.)

After the entire group of .DEFI pseudo-ops should follow a single .CREATI pseudo-op. This causes the instruction table as it has been built to be written as a disk file such that it may be recalled during future assemblies with the .DEFLTI pseudo-op.

It has proven most favorable in the past to guarantee that the file whose ddname is ARCFILE (see section B.1 above) is empty while generating the instruction table. This has been accomplished by temporarily renaming the file assigned to ARCFILE for the duration of the assembly generating the instruction table, so that the file assigned to ARCFILE is a dummy file.

B.3 Defining and Creating the Architecture File

The cross-assembler as it exists allows a great deal of flexibility to the user. Because of this, at each assembly time the assembler must receive the metalanguage commands defining the machine for which the cross-assembly is to take place, the canonic classes of instructions, allowable operands for a class, etc. Rather than require these definitions at the beginning of all programs to be assembled, the option has been created to process first another file. This other file (whose ddname in the EXEC file is ARCFILE) is assembled just as any other source file, with the exception that no listing file will be produced for any source statements appearing in this file. By using this architecture file the requirements produced by the generalization of the cross-assembler are made transparent to the casual user.

It is important to remember that each time the cross-assembler is loaded it starts cold, with no storage areas and with an instruction table containing only the two pseudo-ops .DARCH and .DEFI. Therefore clearly the first instruction of an architecture file should be the .DARCH pseudo-op which causes the cross-assembler to generate the necessary storage areas. The next thing that should be done is to define the .DEFLTI pseudo-op by using the

.DEFI pseudo-op. This is done with the statement:

```
.DEFI    .DEFLTI = '0,0,0,0,0,1,0,4
```

As soon as this definition is given a .DEFLTI should be issued to read the default instruction table from disk (which defines the remaining instruction mnemonics). Once this has been accomplished the remaining structure of the assembler should be defined.

Perhaps the best way to continue at this point is to define the instruction canonic classes. This task is accomplished by using the metalanguage commands .DCLASS and .DOP, a single .DCLASS required for each canonic class and a .DOP required for each allowable operand of that class. The .DCLASS contains such information as canonic class number, minimum and maximum number of operands which may be coded for an occurrence of an instruction of this class, the number of .DOP's coded for this class, and a special number which may be used by the cross-assembler to indicate a required operand which is missing (for a complete definition of .DCLASS and .DOP see Section C). The .DOP is used to specify an operand mnemonic and the appropriate bits to be set by an appearance of this operand, it associates this operand with the appropriate canonic class, specifies the position in which this operand may appear, and optionally specifies an IC type for instructions requiring use of the large IC field.

Sometimes an exclusivity set may exist for which multiple members may appear in the same memory location as long as a certain field is the same for all instructions. For example, consider the case where multiple shifters use the same bit(s) to determine which type of shifting should occur. In

this case shifts may be done in multiple shifters as long as the same type of shifting is attempted. The cross-assembler may be informed that such a situation exists for a certain exclusivity set by using the define exclusivity set check pseudo-op, .DEXCLC.

The translation tables for immediate constants, if any are required, should then be defined. This requires a single .DTRANS to allocate storage for the translation tables, and at least one .DTTAB pseudo-op for each of the translation tables to specify table numbers. It is important that the .DTRANS pseudo-op precede any .DTTAB, since it defines the translation table environment and allocates translation table storage.

Once any necessary translation tables have been defined the immediate constant classes should be defined using the .DEFIC pseudo-op. It is important to remember that all forward referenced variables will be assumed to be members of IC class 1.

The architecture file may be terminated by optionally changing either the default input or the default output base. The default input base is originally decimal, but may be changed with the .DIBASE pseudo-op to either decimal, binary, octal, or hexadecimal. It specifies the base of a number not preceded by a base indicator (the base indicators are .B, .Q, .D, .H, or '). The default output base is merely the base used when creating the address and object fields of the listing. The original default output base is octal, although the .DOBASE pseudo-op may be used to change the output base to either octal or hexadecimal.

This should complete the architecture file. Note that the architecture file should contain solely metalanguage commands, i.e., it should not contain any instructions which produce executable code.

B.4 Assembling a Program

Once all of the previous steps have been completed assembly is indeed an easy task. The user must merely enter the name of the EXEC file, followed by the filename and filetype of the source file to be assembled. For example, if an EXEC file named PMPASM EXEC has been created, and a user wishes to assemble a program named MYPROG FORTRAN he would merely need to enter the command:

```
PMPASM MYPROG FORTRAN
```

Assuming the EXEC file has been created in the standard way discussed, this would produce three output disk files; namely MYPROG LISTING, MYPROG OBJECT, and GENADR DATA. The listing file may be either printed offline or examined from the terminal. Typically another program is required which reads the OBJECT file and GENADR DATA, formats the object binary, and produces some type of output on a medium which may be input by the machine on which the cross-assembled program is to execute. (Typically this might be paper tape.)

The pseudo-ops required in a user's program will be minimal. Both the .DEFC and .LOC may be observed frequently, as will be the case for members of user-defined classes 15 and 16. The pseudo-ops .DEFI, .DIBASE, and .DOBASE might also appear in a user program but with a low frequency. The remaining pseudo-ops should not appear in a user program, since these necessary meta-language definitions may be more easily given in the architecture file.

As a final note, it should be mentioned that the cross-assembler anticipates reading 80-character records, but only the first 72 characters are examined. This allows the user the flexibility of using standard sequencing from the CMS text editor.

C. Pseudo-op Definitions

The following section presents a detailed description of the individual pseudo-ops recognized by the cross-assembler. When applicable special requirements for individual pseudo-ops are also listed.

Pseudo-op: .DARCH

Syntax: .DARCH numbits,lowmem,memsize,numclass,numiclass

Purpose: The .DARCH pseudo-op defines the word size, number of program memory words available, number of instruction canonic classes, and the number of classes for an immediate constant. Since allocation of most storage areas is done dynamically when this pseudo-op is processed it MUST be the first line read by the cross-assembler.

Operands: The operand numbits specifies the number of bits per instruction word, with an upper limit of 128 bits. The operands lowmem and memsize specify the lower and upper addresses respectively of the section of program memory to be generated by the cross-assembler. The number of canonic classes of instructions is specified by numclass, and the number of immediate constant classes is specified by numiclass.

Pseudo-op: .DCLASS

Syntax: .DCLASS classnum,minops,maxops,numops,requiredops

Purpose: The .DCLASS pseudo-op is used to define the properties of a canonic class of instructions.

Operands: The canonic class classnum will consist of instructions which must have at least minops but no more than maxops operands coded. The operand numops should be some number less than or equal to 10 which tells the cross-assembler how many operand choices are to be given for this class by the .DOP pseudo-op.

The last operand, requiredops, permits the cross-assembler to flag an instruction which may have the appropriate number of operands coded, but is missing an essential operand. For example, let us consider the READ instruction of PMP-1, which takes the form:

```
READ  IC,[B1, B2, O]
```

where [B1, B2, O] indicates one or more of B1, B2 or O may be coded. This instruction reads the contents of PE RAM location IC (where IC specifies some immediate constant) and puts the result into one or more of the destination registers B1, B2, and O. The immediate constant is a required operand and therefore the cross-assembler should flag the line:

since even though it contains the appropriate number of operands an essential operand is missing. If the bit of requiredops associated with the operand IC is set = 1 the cross-assembler will be able to flag the erroneous line above.

requiredops specifies a 15-bit structure with each bit = 1 if the corresponding operand (or one of the corresponding group of operands) must be coded, or a bit = 0 if its corresponding operand is not required. The first 10 bits are assigned directly to the 10 possible operands as defined by the .DOP pseudo-op. Bit 1 (MSB) is associated with the operand to be indexed as the first operand in the list, bit 2 with operand 2, etc. In the case of the READ instruction above, if the operands are defined such that IC is indexed as operand 1, B1 as operand 2, B2 as operand 3, and 0 as operand 4, then bit 1 (MSB) of requiredops should be =1. That is, requiredops should be 40000_8. Bits 11-15 are associated with groups of operands and should be set = 1 only if one member of that group must be coded as an operand. The bits are associated as follows:

Bit Number	Associated Operand Group
11	1 or 2
12	1, 2, or 3
13	2 or 3
14	2, 3, or 4
15	1, 2, 3, or 4

As an example of a situation in which one of the bits 11-15 of requiredops would be used consider the ADD instruction of PMP-I. It is of the form:

ADD {Y,IC}, [X,S,EO]

where the first operand must be either Y or IC, followed by one or more operands chosen from X, S, or EO. Let us consider that when the operands are defined they are indexed such that Y is indexed as operand 1, IC as operand 2, X as operand 3, S as operand 4, and EO as operand 5. Then setting bit 11 of requiredops = 1 (i.e. requiredops = 20_8) for the instruction class containing ADD will cause the cross-assembler to verify that at least one of Y or IC was coded, thus flagging the invalid line:

ADD X,S

Pseudo-op: .DOP

Syntax: .DOP operand,bitrep,class,numinclass,numopcoded,ictype

Purpose: The .DOP pseudo-op is used to define an operand, associating it with a canonic class, and specifying the action to be taken when the defined operand is coded.

Operands: The first operand, operand, is the character representation of the operand. Only the first four characters will be recognized. If an operand is to be an immediate constant this field must be the 2 characters 'IC'.

bitrep must be an octal constant beginning with an apostrophe. It specifies a word which will be logically OR'ed with the instruction word at assembly time, thereby indicating which bits should be set = 1 when operand appears. If the binary string generated from bitrep is shorter than an instruction word it will be left justified and padded on the right with zeroes. Even if the operand is an IC, bitrep will be logically OR'ed with the instruction word.

class is the canonic class number of the canonic class of instructions being defined. numinclass is the element number of this particular element of the operand set associated with canonic class class. The cross-assembler merely uses numinclass as an index to the operand set, with members of the set examined from low index to high index.

numopcoded may be either the position number of this operand if its order in the coded operand list is essential, or zero if it may be coded in any position. The final operand ictype is optional and is only coded if the operand is an immediate constant (i.e., operand = IC). ictype specifies the class number of the IC class to which this immediate constant belongs. If ictype = 1 is coded then the ictype as specified in the individual instruction definition (.DEFI) will be used for lower and upper limits.

Special Notes:

1. Each operand as it is defined via the .DOP pseudo-op becomes a reserved mnemonic and therefore may not be used as a label.
2. If the operand is 'IC' then numopcoded should never be zero.
3. The cross-assembler searches the set of available operands starting with that operand with numinclass = 1 and using numinclass as an index through the set. Therefore if an operand may be either an IC or something else the other possibilities should have a lower numinclass than 'IC' in order to prevent the cross-assembler from attempting to evaluate a reserved mnemonic as an immediate constant.

Pseudo-op: .DEFLTI

Syntax: .DEFLTI

Purpose: This pseudo-op retrieves the default instruction table from the disk. It causes the disk file defined by a FILEDEF command in the EXEC file with a ddname of INSFILE to be read. This allows the user to obtain the predefined instruction set.

Pseudo-op: .CREATI

Syntax: .CREATI

Purpose: The .CREATI pseudo-op is used to create a disk instruction file by writing the current instruction table to a disk file. The filename is defined by a FILEDEF in the EXEC file with a ddname of INSFILE. This pseudo-op is used when initially defining an instruction set, or may be used to add more mnemonics to a predefined instruction set.

Pseudo-op: .DEFI

Syntax: .DEFI inst=bits,excl,ictype,icreq,fref,pseudo,xmine,type

Purpose: The .DEFI is used to define an instruction mnemonic. This instruction mnemonic is added to the instruction table for the current assembly only unless a .CREATI pseudo-op follows before completion of assembly.

Operands: The mnemonic itself is given as inst. This operand should be a character string, with only the first seven characters examined by the cross-assembler. The operand bits specifies the instruction word bits which are set to 1 by the instruction. bits must be an octal number, must be preceded by an apostrophe, and may not be any longer than dictated by the number of bits in an instruction word. The cross-assembler will, however, accept a shorter-than-necessary string which it left-justifies with zero fill on the right.

The operand excl specifies a 10-bit structure with each bit representing an exclusivity set. (Instructions which are mutually exclusive, i.e. no more than 1 may be coded per line, belong to the same exclusivity set. An instruction may belong to more than 1 exclusivity set.)

The operand ictype specifies the IC class number if the instruction uses the large IC field, or 0 if this field is not used. This field is used whenever the class of an IC is specified by the .DOP to be = 1. icreq should be = 1 if the large IC field is required for this instruction, =0 if not. fref should be =1 if forward referencing is allowable, =0 if not. pseudo should be =1 if the instruction is a pseudo-op, =0 if not.

xmine should be =1 if the operands coded for an instruction should be examined with appropriate bits being set even if an insufficient number of operands were coded, and xmine should be =0 if coding an insufficient number of operands should cause no operands to be examined. Finally, the operand type specifies the canonic class number of the instructions (or pseudo-op class for pseudo-ops).

Pseudo-op: .DTRANS

Syntax: .DTRANS numtrans,sizetrans

Purpose: The pseudo-op .DTRANS is used to define the translation tables optionally used by an immediate constant class. It defines the number of translation tables needed as well as the size of the largest table. This pseudo-op must precede any attempts to specify translation table elements with the .DTTAB pseudo-op.

Operands: The total number of translation tables necessary is given by numtrans. The second operand, sizetrans, specifies the number of elements in the largest translation table.

Pseudo-op: .DTTAB

Syntax: .DTTAB num,offset,begindex,value1,value2,...

Purpose: The .DTTAB pseudo-op is used to specify elements of an immediate constant translation table.

Operands: The first operand, num, specifies the number of the translation table. offset is the offset amount which must be added to an immediate constant to convert this to a table index from 1 to n, where n is the number of elements in the table. The operand begindex merely specifies which table entry follows, and value1, value2, etc. are the actual translation table entries (with value1 being the begindexth table entry).

Pseudo-op: .DEXCLC

Syntax: .DEXCLC exclset,beginbit,endbit

Purpose: The .DEXCLC pseudo-op defines an exclusivity set check which may allow more than one member of an exclusivity set to coexist on the same instruction line. It is used for exclusivity sets where a particular field is used by all members of this set, but multiple members may coexist if this field is to be the same for all members.

Operands: The exclusivity set number (from 1 to 10) is given as the first operand, exclset. The bit numbers corresponding to the begin and end of the field used by both instructions are specified respectively by the operands beginbit and endbit.

Pseudo-op: .DEFIC

Syntax: .DEFIC class,lowlimit,hilimit,begin,end,trantab

Purpose: This pseudo-op specifies upper and lower limits for an immediate constant class, the begin and end bit positions of the instruction word for an IC of this particular class, and a translation table number if the immediate constant to be coded requires some translation before it is put in the instruction word.

Operands: The operand class specifies which IC class is being defined. lowlimit and hilimit define respectively the lower and upper values which an IC of this class may attain. begin and end specify begin and end bit positions of the instruction word to be used by this IC class (with the MSB being called bit 1), and trantab either is the number of a translation table necessary to convert an IC coded to a value placed in the instruction word, or 0 if no such translation is necessary.

Note: IC class 1 should always define the "large IC field", since class 1 limits are used for resolving forward references as they are defined.

Pseudo-op: .DIBASE

Syntax: .DIBASE base

Purpose: The .DIBASE pseudo-op may be used to define the default input base, that is, the base assumed for any numerical immediate constant not preceded by a base indicator.

Operand: The single operand base should be either 2, 8, 10, or 16 (assuming the previous default input base was 10).

Pseudo-op: .DOBASE

Syntax: .DOBASE base

Purpose: The .DOBASE pseudo-op defines the output base used in printing the address and object columns of the listing.

Operand: The single operand base may be only 8 or 16.

Pseudo-op: .LOC

Syntax: .LOC value

Purpose: The .LOC pseudo-op is used to set the address location counter. It is not allowable to set this counter to a value less than the current value.

Operand: The single operand value should be the value to which the address counter is to be set.

Pseudo-op: .DEFC

Syntax: .DEFC name=value

Purpose: The .DEFC pseudo-op is used to assign a value to a given mnemonic label.

Operand: The operand name is the mnemonic label to be defined. Only the first seven characters are recognized by the cross-assembler. The value assigned is given by the operand value, which may be any predefined label or any constant.

Pseudo-op: User-defined Pseudo-op Class 15

Syntax: 1 mnemonic (opcode with no operands)

Purpose: This pseudo-op class allows the user to define mnemonics that allow a certain bit (or bits) to be set =1 in all subsequent instruction words.

Example: Assume a user is dealing with a 24-bit machine where bit 14 (with MSB = bit 1) is a mask bit for interrupts, i.e., no interrupts are allowed when bit 14 = 1, interrupts are allowed when bit 14 = 0. The user is able to define a pseudo-op, say MSK, which will cause the mask bit to be set to 1 in all subsequently assembled instruction words until another instruction (of user-defined pseudo-op class 16) turns this bit off again. To define the instruction MSK, the user would code:

```
.DEFI MSK='00002,0,0,0,0,1,0,15
```

Pseudo-op: User-defined Pseudo-op Class 16

Syntax: 1 mnemonic (opcode with no operands)

Purpose: This pseudo-op class allows the user to define mnemonics that allow a bit (or bits) set = 1 by an instruction of pseudo-op class 15 to be cleared again (=0) in all subsequent instruction words.

Example: Let us again consider the example discussed under "user-defined Pseudo-op Class 15". After the user has completed the section of code where bit 14 = 1 (i.e.,

interrupts were disabled) he wishes the assembler to again produce object code with bit 14=0 (i.e., interrupts should be enabled again). To accomplish this a pseudo-op, say UNMSK, may be defined which will undo the work of the MSK pseudo-op. To define the pseudo-op UNMSK the user would code:

```
.DEFI UNMSK='00002,0,0,0,0,1,0,16
```

D. Outputs from the General Cross-Assembler

The cross-assembler produces several outputs including a listing file, a file designed for the terminal (consisting of error messages and statements producing these errors), and a binary object file with an address file to serve as an index. Assuming the listing file has been written to disk it may be either printed offline or typed at a terminal, as the user may desire. The file intended for the terminal is to give the user an indication of the success of the assembly, and there is usually no reason to save this file.

The binary object file usually will require another follow-up program to read, format, and transmit the data to some medium so that it may be acceptably input to the user's machine. It is this binary object file and associated address file upon which we shall focus our attention in this section.

Only the program memory locations for which an instruction was coded will be written to the binary object file, thus creating the need for the associated address file. The address file consists of pairs of begin and end addresses.

If an end address is one less than the begin address this means that there were no binary object words written for this pair, and it may be ignored. Since only the .LOC, pseudo-op allows an address change of more than one location, this only occurs if the source program begins with a .LOC, or if two .LOC pseudo-ops are coded consecutively with no intervening instructions. (What actually happens is that when a .LOC is encountered the cross-assembler decrements its address counter which points to the address of the next available program memory word, writes this value to the address file, then sets its address counter to the value of the operand of the .LOC and writes the value to the address file.) The address file is terminated by a pair of begin and end addresses, both of which are 999999.

The address file is created using record-oriented transmission in PL/I. Assuming ADR_CNT is a 31-bit binary fixed variable, then a typical output statement to the address file might be:

```
WRITE FILE (ADRFILE) FROM (ADR_CNT);
```

Thus a PL/I input statement should also read into a 31-bit binary fixed variable, say ADR_INDEX with a statement similar to:

```
READ FILE (ADRFILE) INTO (ADR_INDEX);
```

As previously mentioned in section B.1 the address file is created with ddname ADRFILE having a blocksize =80 and variable blocked record format (i.e., in the FILEDEF use BLOCK 80 RECFM VB).

The binary object file is created using PL/I stream-oriented transmission. Each memory location is written with a B-format (bit-string format) of length NUMBITS where NUMBITS stands for the number of bits in the user's machine. Thus assuming the binary object is stored in an array called BIN_COD and ADR_CNT is a 31-bit binary fixed variable, a typical PL/I output statement to the file BINFILE might look like:

```
PUT FILE (BINFILE) EDIT (BIN_COD (ADR_CNT))(B(NUMBITS));
```

Note that using a record length of 80 and blocksize 800 (i.e., in the FILEDEF using LRECL 80 BLOCK 800 RECFM FB) exactly one program memory location per record is written only for a machine with an 80-bit program memory word. That is, a single output record does not necessarily correspond to a single program memory location.

II. Example of Creating a Tailored Version of the Cross-Assembler for PMP-I

This section is devoted to a discussion of a specific example of using the general cross-assembler for the Parallel Microprogrammable Processor (or PMP-I) built at Lincoln Laboratory. It is also intended to serve as a user's guide for those wishing to use the cross-assembler to assemble PMP-I code.

Perhaps a brief word describing the architecture of PMP-I is in order. Conceptually a PMP consists of a control unit and a number of identical processing modules (PM's), each containing a processor element (PE) and a data memory. The sole PMP-I in existence contains a single PE, but if multiple PE's are present all perform the same operation simultaneously.

Both the controller and PE's have a 24-bit architecture. The actual data-processing tasks are performed in the PE(s), while the controller sequences through the program, computes addresses for PE RAM(s) and PM data memories, and communicates with external devices.

A 60-bit program memory word consists of 38 PE control bits, a 12-bit immediate constant (IC) field, and a 10-bit controller field. The 10-bit controller field may be further subdivided into an 8-bit field specifying which of 256 controller operations is to take place, a bit to specify whether or not the instruction pipeline is inhibited on a branch instruction, and a bit to specify the source for the PE RAM address used. For completeness, a diagram of the PE architecture appears as Appendix L, and a diagram of the 60-bit program memory word appears in Appendix D.

Step 1 in the procedure of using the cross-assembler for a new machine is to group the instruction mnemonics into canonic classes, each canonic class consisting of members having the same operand set. The canonic class divisions for PMP-I are given in Appendix B. At a first glance, it may appear that canonic classes 11 and 12 (the B1- and B2-shift instructions respectively) have the same operand set as canonic class 2, namely all three classes require a single immediate constant (IC) as an operand. However upon more careful inspection it is seen that the IC in canonic class 2 is intended for program memory bits 49-60, while the IC in canonic class 11 (B1-shift instructions) affects the B1-shift bits 29-30 and the IC in canonic class 12 (B2-shift instructions) affects the B2-shift bits 35-36. Thus criteria for forming canonic classes should be that all members of a class

should have the same operand set, and if that operand set includes an IC that IC must be intended for the same position in the program memory word.

A somewhat easier task is determining the IC classes. This is accomplished by creating a list of all possible bit positions for which an IC is intended, and for each of these groups deciding the limits that may be placed on an IC. For PMP-I, there are 3 possible locations in a program memory word for which an IC is intended, namely the large IC field (bits 49-60), the B1-shift bits (bits 29-30), and the B2-shift bits (bits 35-36). Furthermore, on the first of these groups it is convenient to place 5 limit restrictions, namely -2048 to 2047, 0 to 2047, 0 to 1023, 0 to 4095, and 0 to 4. Thus the 7 IC classes as listed in Appendix C were created. (Note that a translation table is convenient for the two B-shift classes in order to allow a programmer to code an IC of 1-4 indicating the number of bits to shift, but allows the cross-assembler to produce the code as listed in the bit definitions of Appendix D.)

Determining the exclusivity sets requires careful examination of the program memory bit assignments. Exclusivity sets are useful to indicate instructions which are mutually exclusive (i.e., may not appear in the same program memory word) but do not try to set the same bits = 1. The cross-assembler performs a logical AND as each instruction is added to a program memory word, producing an error message if a non-zero result is obtained. Thus, exclusivity sets should consist of instructions requiring the same program memory word bits but may not necessarily produce a conflict message from this logical ANDing.

Three immediate exclusivity set candidates for PMP-I are ALU1 operations (bits 1-6), ALU2 operations (bits 7-12), and control opcodes (bits 41-48). More careful consideration indicates an exclusivity set for the instructions that use the recirculating bus (bit 15) will indicate a conflict if an erroneous attempt is made to set the recirculating bus to the contents of the M register for clocking into one of the B registers, and to the contents of a PE RAM location for clocking into the other B register. This eliminates a problem with the first version of the cross-assembler for PMP-I which did not flag the erroneous combination of the READ and MOVM instructions.

Examining bits 31, 32 indicates that these bits control the shifting that may occur either in B1 or B2. Thus B1 and B2 may both perform shifts simultaneously as long as they are the same type of shift. This adds an extra complexity: the B-shift instructions are mutually exclusive only if they attempt to set bits 31, 32 differently. This problem is overcome by defining an exclusivity set for the shift instructions as well as an exclusivity set check (via .DEXCLC) which examines bits 31, 32 and produces no error message if these settings are the same. The final exclusivity set is defined for PE RAM access instructions, to cover the case of an attempt to read and write the RAM simultaneously where either the read or write (but not both) use the contents of the S-register (PEAS=1) as its effective address.

These 6 exclusivity sets complete those defined for PMP-I. Perhaps it may not be immediately obvious why other classes were not defined. For example, one might notice that an attempt to perform a logical right shift of

4 bits as well as a logical right shift of 2 bits in B1 would not produce an exclusivity set conflict. However both of these instructions would attempt to set bit 28=1 and the cross-assembler would detect this as a conflict when it logically ANDed the two bit strings of the instructions.

Once the language of the machine has been conceptualized in this manner the user is ready to begin creating his definition files as indicated below.

A. Setting Up the EXEC for PMP-I

The EXEC file used for assembling PMP-I code was named PMPASM EXEC and appears as Appendix F. The first four lines are standard and appear as given in Part I, Section B.1. The FILEDEFS for ddnames SFILE, ADRFILE, WFILE, BINFILE, and TFILE are also the standard forms as given in Part I, Section B.1.

The name chosen for the architecture file was PMP ARCH, and thus the FILDEF for ARCFILE is:

```
FI ARCFILE DISK PMP ARCH (LRECL 80 BLOCK 800 RECFM FB
```

Similarly the name chosen for the disk instruction file was PMPINS DATA, and thus the FILEDEF for INSFILE is:

```
FI INSFILE DISK PMPINS DATA (LRECL 80 BLOCK 800 RECFM FB
```

B. Generating an Instruction Table for PMP-I

The PMP-I instruction table was created by an assembly of the file PMPINST FORTRAN. This file, which appears as Appendix H, begins with the .DARCH pseudo-op for PMP-I which must declare PMP-I as a 60-bit machine,

having 12 instruction canonic classes and 7 IC classes. The lower and upper memory limits are not particularly relevant, since PMPINST FORTRAN will contain no source code which will produce executable code.

The next 12 statements are the required .DEFI statements to define the pseudo-ops not built-in to the cross-assembler. This includes all pseudo-ops except .DARCH and .DEFI, and these definitions should be present in all files generating an instruction table.

The remainder of PMPINST contains a .DEFI pseudo-op to define each of the mnemonics given in Reference 2. Each .DEFI is created from the definition (as given in Reference 2), memory bit assignments (Appendix D), exclusivity set definitions (Appendix E), IC class table (Appendix C), canonic class table (Appendix B), and a list of control opcodes (Reference 2, Appendix A). As an example, let us consider the mnemonic BMPY.

This instruction requires an addition (F=A PLUS B) in both ALUs, clocking of both A1 and A2 if the MSB of the M-register is non-zero, and logically left-shifting the contents of the M-register one bit. From the ALU function select table (Appendix K) we see that for the function F=A PLUS B we must choose the ALU control lines S_0, S_1, S_2, S_3, C_n, and M such that S_2, S_1, M=0 and S_3, S_0, and C_n=1. Since the addition is to be performed in both ALUs, this means bits 1, 4, 5, 7, 10, and 11 should be set =1. Furthermore, examining the memory bit assignments given in Appendix D indicates that M is logically left-shifted one bit if bit 18=1, and A1, A2 will be clocked when the MSB of M is 1 if bits 22, 23, 24 are given as 011.

Thus, the 60-bit word for BMPY may be given (in octal) by: 46460103000000000000. BMPY requires the use of both ALU1 and ALU2, thereby belonging to exclusivity sets 9 and 10. Since these are the two LSB's of the exclusivity set field, the exclusivity number coded is 3. Since BMPY does not require (nor allow) any IC, the IC class is coded as 0, and both flags used to indicate a required IC and when forward referencing is allowed are also 0. BMPY is not a pseudo-op therefore the pseudo-op flag should be 0, and since no operands are allowed the flag to indicate whether to examine any operands if an insufficient number were coded is not applicable (and therefore coded as 0). From the canonic class table we find BMPY belongs to canonic class 1. Thus the definition line for BMPY becomes:

```
.DEFI   BMPY='46460103,3,0,0,0,0,0,1
```

When all instruction mnemonics have been similarly defined and stored as part of the instruction table it is important to write this table to disk. This is accomplished by use of the pseudo-op .CREATI.

As previously mentioned, if the cross-assembler is used to assemble this file the appropriate disk instruction file will be created. It is important to remember however that no architecture file should be read beforehand, and so when assembling PMPINST FORTRAN the file PMP ARCH should be renamed (often times TEMP ARCH has been used).

C. Creating an Architecture File for PMP-I

The EXEC file for PMP-I cross-assemblies names a file called PMP ARCH to be assembled before assembling a user source program. Thus PMP ARCH (which appears as Appendix G) may be used to define the particulars of PMP-I.

Naturally the first statement of PMP ARCH must be a .DARCH, since it is this pseudo-op which causes the cross-assembler to dynamically allocate necessary storage areas. The operands of .DARCH are clear, since PMP-I is a 60-bit machine with 2K of program memory, and for which the instructions have been divided into 12 canonic classes and the IC's into 7 IC classes. Immediately following comes the definition of the pseudo-op .DEFLTI and by issuing the .DEFLTI pseudo-op the default instruction table previously created from PMPINST FORTRAN is retrieved from disk.

The 12 canonic classes are then defined, each class definition consisting of one .DCLASS pseudo-op and a .DOP pseudo-op for each allowable operand. (Notice that canonic class 1 allows no operands and therefore no .DOP is used.) Special attention is drawn to the fact that the fourth operand of .DOP actually specifies the order in which the cross-assembler examines the list of available operands. In particular, this implies that when an operand may be either an IC or something else the other possibilities should be checked for first, otherwise the cross-assembler will incorrectly attempt to evaluate an operand which is not an IC as an IC.

As an example let us consider the definition of canonic class 9. From the canonic class table appearing as Appendix B we see that a class 9 instruction may have 2, 3, or 4 operands, the first of which must be either an

immediate constant (IC) or Y, and the remaining operands should be one or more from the set {X, S, EO}. Therefore, the .DCLASS pseudo-op for canonic class 9 specifies at least 2 operands but no more than 4 operands may be coded, and 5 operands are included in the operand set. The final operand which specifies any required operands is given as '20. This means that of the 15-bit field corresponding to this operand bit 11 is =1, i.e., of the operands whose indices are 1 and 2 at least one must be coded.

The operands of canonic class 9, in the order of their indices, are Y, IC, X, S, and EO. Notice that the last operand of .DCLASS specifies that either Y or IC must be the first operand coded, which satisfies the canonic class 9 rule. Also, Y must have a lower index than IC (1 vs 2) in order to prevent the cross-assembler from an attempt to evaluate Y as an immediate constant. Both Y and IC must appear as the first operand if they appear and so the fifth operand of .DOP (numopcoded) should be set = 1. The order of X, S, and/or EO is not important and therefore numopcoded is given as zero in these cases.

There is only one exclusivity set check and override case in PMP-I, the shift instructions (exclusivity set 6). The .DEXCLC pseudo-op is used to indicate that if two instructions of exclusivity set 6 are coded for the same program memory word there is no conflict if the shift bits 31-32 are similar for both instructions.

Only one translation table is necessary, the translation table for the shift instructions. The .DTRANS pseudo-op allocates 1 translation table with 4 entries, and the .DTTAB pseudo-op is used to fill this table with the

results to be used for the shift-count-bits (bits 29-30 and 35-36) as indicated in Appendix D.

D. Assembling a User Source Program

After the creation of PMPASM EXEC, PMP ARCH, and PMPINST FORTRAN's assembly to create the instruction table on disk, the burden is removed from the user. The user need only invoke the PMPASM EXEC by typing the CMS command:

PMPASM name type

where name and type are respectively the filename and filetype of the user's source program. All appropriate machine definitions occur automatically, transparent to the user's program.

A follow-up program has also been generated to create a paper tape which may serve as input to PMP-I. This program may be invoked by entering the command:

PMPPUN name

where name is the same filename as entered in the assembly process. Eventually a more sophisticated method will exist whereby the user may send the object binary directly from the IBM 370 to a minicomputer which in turn communicates with PMP-I, but this system does not yet exist.

References

1. C. E. Muehe (20 February 1975), private communication.
2. B. G. Laird (22 February 1977), private communication.
3. P. R. Kretz (2 October 1975), private communication.
4. P. R. Kretz (26 March 1976), private communication.

Appendix A. Definition of Terms and Concepts

Following is a discussion of terms and concepts which appear frequently in the text of this report.

1. Canonic class. Perhaps one of the most fundamental ideas behind the use of the general cross-assembler is defining canonic classes and dividing the instructions into the appropriate canonic classes. A canonic class is formed by a group of instructions, all of which have the same list of allowable operands. The sole difference between two canonic classes is that some difference exists either in allowable number of operands, allowable operands, or action taken by the cross-assembler (e.g., in the case of PMP-I, LRSB1 and LRSB2 are not in same class since in the first case the cross-assembler defines B1-shift bits and in the second case B2-shift bits are defined).

2. Cross-assembler. An assembler which runs on one machine, but assembles code for another machine.

3. Delimiter. One of a set of special control characters recognized by the cross-assembler. This set consists of the comma (,), semi-colon (;), colon (:), slash (/), equals (=), and blank ( ). Delimiters are used to break the source line into segments (see Token).

4. Exclusivity Set. This concept pertains to two or more instructions which are mutually exclusive, i.e., they may not appear in the same program memory word. The requirement of defining exclusivity sets arises from the design of the cross-assembler for micro-programmable machines, for which more than one instruction may be coded in the same program memory location as long as conflicting sources, destinations, and/or wires are not required.

5. IC (Immediate Constant). The term IC (or Immediate Constant) refers to an operand which is either a numerical constant or a symbolic label (which the cross-assembler converts to a numerical constant).

6. IC Class. An IC class consists of a group of numerical constants, bounded by both an upper and a lower limit. In addition to these limits an IC class has associated with it beginning and ending bit numbers which specify where in an instruction word it is to be placed. Another feature allowed in an IC class is translating the input constant by means of a translation table.

7. Large IC Field. The "large IC field" referred to in the text is the field typically used to hold addresses or address displacements. For example, on the PMP this would be bits 49 through 60 (with MSB = 1); on the Data General Nova this would be bits 8 through 15 (with MSB = 0).

8. Metalanguage. A language used in the definition of other languages.

9. Opcode. An operation code, which specifies operation or instruction to be performed. Each opcode is typically represented by an abbreviation, or mnemonic.

10. Operand. Typically used to specify either sources and/or destinations for an operation code. For example, considering the PMP instruction ADD1 A1,M the opcode is ADD1 and operands (in this case destinations) are A1 and M.

11. PMP. Parallel Microprogrammable Processor. A high-speed processor designed for digital signal processing applications at Massachusetts Institute of Technology, Lincoln Laboratory.

12. Program Counter. The term program counter as used here is synonymous with the address of the current program memory location.

13. Pseudo-op. A command which controls the cross-assembler, but itself produces no executable object code.

14. Token. A character string separated from the remainder of a source line by delimiters. A token may contain no imbedded delimiters. A token will be either a label, opcode, or operand.

15. Translation Table. Used to convert the actual IC coded by a programmer to the bit string placed in a program memory word, if the two are not equivalent. An optional offset may also be specified, which at assembly time is added to the coded IC before using that number as the index to the appropriate translation table index.

Appendix B. Canonic Classes For PMP-I μ-Instructions

Class	Operand Description	Mnemonics in Class
1	No operands, ever.	PE(BMPY, LLSM, FFB2, CAB, CMO, BMPY1, BMPY2, BDIV, BFIX, BFLO, WRS) C(JMPX, JMKX, MOVSY, LLSY4, DREADY, DREADXY, NXTREAD, ENABL, DSABL, PSAVS, LDEI, LDEIB, HALT, SETAUX, CLRAUX, PINTS, SWAPSY, MOVSYXS)
2	One operand, an immediate constant, which must be coded.	PE(MOVI,WR) C(LLS4,LDX,DREADX,LDEO, SETIO,CLRIO, LDPM 0≤IC≤1023 -2048≤IC≤2047 0≤IC≤4095 1≤IC≤4 JMP, JMK, JMPXL, JMKXL, JMPXE, JMKXE, JMPXG, JMKXG, JMPAL, JMKAL, JMPAE, JMKAE, JMPAG, JMKAG, JMPTX, JMKTX, JMPDEI, JMKDEI, JMPDEO, JMKDEO, JMPDHP, JMKDHP, JMPXNE, JMKXNE, JMPXLE, JMKXLE, JMPXGE, JMKXGE, JMPOVF, JMKOVF) 0≤IC≤2047
3	0, 1, or 2 operands. If 0, instruction behaves as a no-op and is flagged. Operands to be chosen from the set {A1,M}.	PE(LAB, SAB, MOVA1, MOVB1, COMA1, COMB1, OR1, AND1, XOR1, ADD1, SUB1, LLSA1, ZRO1, ONES1)

Class	Operand Description	Mnemonics in Class	
4	0, 1 or 2 operands. If 0, instruction behaves as a no-op and is flagged. Operands to be chosen from {A2,M}.	PE(MOVA2, MOVB2, COMA2, COMB2, OR2, AND2, XOR2, ADD2, SUB2, LLSA2, ZRO2, ONES2, ABSB)	
5	1, 2, or 3 operands. At least 1 must be specified. Operands are to be chosen from {X,S,EO}.	C(MOVX, MOVY, INCX, DECX, LLSX, LEASX)	
6	2, 3, or 4 operands. At least 2 must be specified. First operand must be an immediate constant, followed by one or more of the set {B1,B2,0}.	PE(READ)	$0 \leq IC \leq 1023$
7	1 operand, either an immediate constant or Y, which must be specified	C(COMP,	MEMX, MEMY, STAS MSKIO, SETMUX)
		$-2048 \leq IC \leq 2047$	$0 \leq IC \leq 1023$ $0 \leq IC \leq 4095$
8	1 operand, either an immediate constant or EI, which must be specified.	C(LDS)	$-2048 \leq IC \leq 2047$
9	2, 3, or 4 operands. At least 2 must be coded. First operand must be either an immediate constant or Y. Remaining operands are chosen from the set {X, S, EO}.	C(ADD, SUB, AND, OR, XOR)	$-2048 \leq IC \leq 2047$

Class	Operand Description	Mnemonics in Class	
10	1, 2, or 3 operands, at least 1 must be coded. Operands are chosen from the set {B1,B2,0}.	PE(MOVM, READS)	
11	1 operand, an immediate constant, which must be coded.	PE(ARSB1, LRSB1, LEARSB1)	$1 \leq IC \leq 4$
12	1 operand, an immediate constant, which must be coded.	PE(ARSB2, LRSB2, LEARSB2)	$1 \leq IC \leq 4$

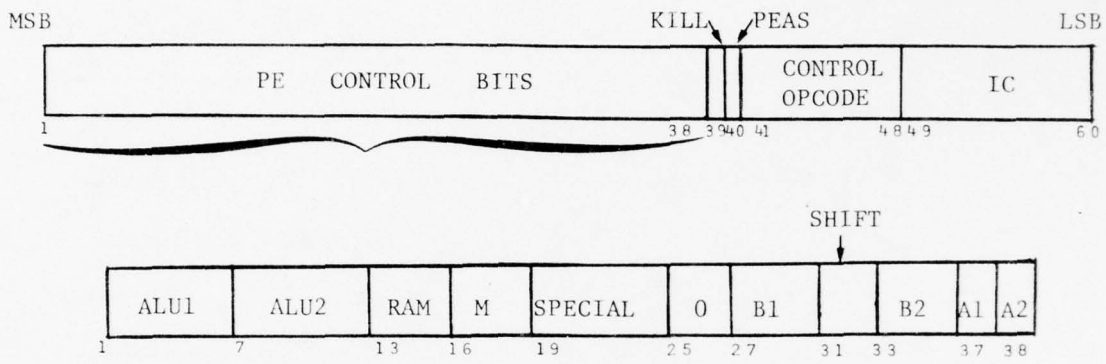
APPENDIX C. IC Classes and Translation Tables for PMP-I

IC Class #	Limits	Beginning Bit #	Ending Bit #	Translation Table #
1	$-2048 \leq IC \leq 2047$	49	60	0(None)
2	$0 \leq IC \leq 2047$	49	60	0(None)
3	$0 \leq IC \leq 1023$	49	60	0(None)
4	$0 \leq IC \leq 4095$	49	60	0(None)
5	$1 \leq IC \leq 4$	49	60	0(None)
6	$1 \leq IC \leq 4$	29	30	1
7	$1 \leq IC \leq 4$	35	36	1

TABLE 1
IC TRANSLATION

Table Entry	Table Value
1	3
2	1
3	2
4	0

APPENDIX D. PMP-I Program Memory Bit Assignments



Note: In the descriptions below, [ ] stands for "contents of register".

Bit	Description of Control Function
1	S_3 for ALU1.
2	S_2 for ALU1.
3	S_1 for ALU1.
4	S_0 for ALU1.
5	C_n for ALU1.
6	M for ALU1.
7	S_3 for ALU2.
8	S_2 for ALU2.
9	S_1 for ALU2.
10	S_0 for ALU2.
11	C_n for ALU2.
12	M for ALU2.

Bit	Description of Control Function
13	Selects input data for RAM. If 1, select [I]. If 0, select [M].
14	Write pulse to PE RAM.
15	Selects data for recirculating bus. If 1, select [M]. If 0, select RAM output.
16	Selects input data for M. If 0, select ALU1 output. If 1, select ALU2 output.
17	Clock M register.
18	Causes data in M to be left-shifted 1 bit. The shift is logical unless control bits 22, 23, 24 are set to '110', in which case the inverted sign bit of ALU1 is shifted in as the LSB.
19	Clock ALU1 sign, overflow, equal, and carry flip-flops.
20	If ALU1 is set for F=A1 and bit 20=1, F=A1 if sign flip-flop = 0, F=B1 if sign flip-flop = 1. (Used to find larger of [A1] and [B1].) If ALU1 is set for F=B1 and bit 20=1, F=B1 if sign flip-flop = 0, F=A1 if sign flip-flop = 1. (Used to find smaller of [A1] and [B1].)
21	Selects sign, overflow, equal, and carry flip-flops for possible clocking into B2.
22,23,24	These special bits are decoded as follows: 001:A2 is clocked if sign bit of M is 1 010:A1 is clocked if sign bit of M is 1 011:A1 and A2 are clocked if sign bit of M is 1 100:Operation in ALU2 is changed if F=A+B is chosen so that F=B if sign of [B2]=0, F=MINUS B if sign of [B2]=1.

Bit	Description of Control Function
22,23,24	101:A1 and A2 are clocked if the sign bit of ALU1 is 0. 110:The sign bit of ALU1 is inverted and clocked into LSB of M if bit 18=1. A1 is clocked if sign bit of ALU1 is 0. 111:A2 is clocked if sign of M is 0. M is logically left-shifted 1 bit unless sign of [M] = 1.
25	The 0-register is unconditionally loaded (based on bit 15), and an external interrupt is generated.
26	If sign of ALU1 is 1; 0-register is loaded (based on bit 15), busy flip-flop is set, and an external interrupt is generated.
27	Selects input for B1. If 1, recirculating bus is chosen. If 0, B1 shifter is chosen.
28	Clocks B1 with the data selected by bit 27.
29,30	Data in B1 is clocked into B1 shifter, and shifted 1,2,3, or 4 places right. Sign bit is filled according to bits 31, 32. Number of shifts is given by bits 29, 30 as follows: 00-shift 4 bits 10-shift 3 bits 01-shift 2 bits 11-shift 1 bit
31,32	These bits determine the type of shifting done in both shifters according to: 00:arithmetic right shifting (Sign bit is shifted into itself and next LSB).

Bit	Description of Control Function
31,32	01:Circular right shifting (LSB becomes MSB). 1X:Logical right shifting (Shift zeroes into MSB).
33	Selects input for B2. If 1, recirculating bus is chosen. If 0, B2 shifter is chosen.
34	Clocks B2 with the data selected by bit 33.
35,36	Data in B2 is clocked into B2 shifter, and shifted 1,2,3, or 4 places right. Sign bit is filled according to bits 31, 32. Number of shifts is controlled by bits 35, 36 as follows: 00-shift 4 bits 10-shift 3 bits 01-shift 2 bits 11-shift 1 bit
37	Clocks A1, loading output of ALU1.
38	Clocks A2, loading output of ALU2.
39	Kill bit. If =1 when a jump is taken, the instruction pipeline is killed.
40	PE address select bit. If =1 the PE RAM address is given by [S]. If 0, PE RAM address is given by the IC field of the program memory word.
41-48	Control opcodes. For a complete listing of the current assignments, see Reference [2] Appendix A.
49-60	Immediate Constant (IC) field.

APPENDIX E. PMP-I Exclusivity Sets

MSB									LSB
BIT	1	2	3	4	5	6	7	8	9	10
					PE RAM ACCESS	B-REG. SHIFT.	RECIRC BUS USED	CONTR. TYPES	ALU2	ALU1

Exclusivity Set	Description
1-4	Not used.
5	Instructions that require a PE RAM access. This includes MOVI, WR, READ, WRS, READS.
6	The B-register shift instructions. Recall that the rule applying to these is that a shift may be done simultaneously in both B-registers only if the same type is done in both. Thus a .DEXCLC (Define Exclusivity Set Check) pseudo-op will be used for the bits which indicate the type of shifting to be done.
7	Instructions using the recirculating bus. This consists of MOVM, READ, and READS.
8	This set consists of all controller instructions, of which only one per program memory word may be coded.
9	All instructions using ALU2.
10	All instructions using ALU1.

APPENDIX F. EXEC File for a PMP-I Cross Assembler

```
FILE: PMPASM   EXEC     A     1/12/77 09:39     M.I.T. LINCOLN LABORATORY

&CONTROL ERROR
CP LINK PLIOPT 191 199 RR
ACCESS 199 Z/Z
GLOBAL TXTLIB SYSLIB PLILIB FORTLIB GRLL
FI ARCFILE DISK PMP ARCH (LRECL 80 BLOCK 800 RECFM FB
FI SFILE DISK &1 &2 &3 (LRECL 80 BLOCK 800 RECFM FB
FI ADRFILE DISK PMPADR DATA (BLOCK 80 RECFM VB
FI WFILE DISK &1 LISTING (LRECL 133 BLOCK 133 RECFM F
FI BINFILE DISK &1 OBJECT (LRECL 80 BLOCK 800 RECFM FB
FI TFILE TERMINAL (LRECL 132
FI INSFILE DISK PMPINS DATA (LRECL 80 BLOCK 800 RECFM FB
LOAD GENASM GENHASH GENBILD (NOMAP NODUP START
```

APPENDIX G. Architecture Definitions for PMP-I

```
FILE: PMP      ARCH      A    12/21/76 09:44     M.I.T. LINCOLN LABORATORY

      .DARCH  60,0,2048,12,7                                          PMP00010
      .DEFI   .DEFLTI='0,0,0,0,0,1,0,4                                PMP00020
      .DEFLTI                                                         PMP00030
      .DCLASS 1,0,0,0,0                                               PMP00040
      .DCLASS 2,1,1,1,1                                               PMP00050
      .DOP    IC,'0,2,1,1,1                                           PMP00060
      .DCLASS 3,1,2,2,0                                               PMP00070
      .DOP    A1,'0000000000004,3,1,0                                 PMP00080
      .DOP    M,'000002,3,2,0                                         PMP00090
      .DCLASS 4,1,2,2,0                                               PMP00100
      .DOP    A2,'0000000000002,4,1,0                                 PMP00110
      .DOP    M,'000006,4,2,0                                         PMP00120
      .DCLASS 5,1,3,3,0                                               PMP00130
      .DOP    X,'0000000000000004,5,1,0                               PMP00140
      .DOP    S,'0000000000000002,5,2,0                               PMP00150
      .DOP    EO,'0000000000000001,5,3,0                              PMP00160
      .DCLASS 6,2,4,4,0                                               PMP00170
      .DOP    IC,'0,6,1,1,3                                           PMP00180
      .DOP    B1,'0000000014,6,2,0                                    PMP00190
      .DOP    B2,'000000000014,6,3,0                                  PMP00200
      .DOP    O,'000000004,6,4,0                                      PMP00210
      .DCLASS 7,1,1,2,0                                               PMP00220
      .DOP    IC,'0,7,2,1,1                                           PMP00230
      .DOP    Y,'0000000000000001,7,1,1                               PMP00240
      .DCLASS 8,1,1,2,0                                               PMP00250
      .DOP    IC,'0,8,2,1,1                                           PMP00260
      .DOP    EI,'0000000000000001,8,1,1                              PMP00270
      .DCLASS 9,2,4,5,'20                                             PMP00280
      .DOP    Y,'0000000000000010,9,1,1                               PMP00290
      .DOP    IC,'0,9,2,1,1                                           PMP00300
      .DOP    X,'0000000000000004,9,3,0                               PMP00310
      .DOP    S,'0000000000000002,9,4,0                               PMP00320
      .DOP    EO,'0000000000000001,9,5,0                              PMP00330
      .DCLASS 10,1,3,3,0                                              PMP00340
      .DOP    B1,'0000000014,10,1,0                                   PMP00350
      .DOP    B2,'000000000014,10,2,0                                 PMP00360
      .DOP    O,'000000004,10,3,0                                     PMP00370
      .DCLASS 11,1,1,1,0                                              PMP00380
      .DOP    IC,'0,11,1,1,6                                          PMP00390
      .DCLASS 12,1,1,1,0                                              PMP00400
      .DOP    IC,'0,12,1,1,7                                          PMP00410
      .DEXCLC 6,31,32                                                 PMP00420
      .DTRANS 1,4                                                     PMP00430
      .DTTAB  1,0,1,3,1,2,0                                           PMP00440
      .DEFIC  1,-2048,2047,49,60,0                                    PMP00450
      .DEFIC  2,0,2047,49,60,0                                        PMP00460
      .DEFIC  3,0,1023,49,60,0                                        PMP00470
      .DEFIC  4,0,4095,49,60,0                                        PMP00480
      .DEFIC  5,1,4,49,60,0                                           PMP00490
      .DEFIC  6,1,4,29,30,1                                           PMP00500
      .DEFIC  7,1,4,35,36,1                                           PMP00510
```

APPENDIX H. Instruction Table File for PMP-I

FILE: PMPINST FORTRAN A 2/08/77 14:29 M.I.T. LINCOLN LABORATORY

```
.DARCH  60,0,2048,12,7
.DEFI   .DEFIC='0,0,0,0,0,1,0,1
.DEFI   .DCLASS='0,0,0,0,0,1,0,3
.DEFI   .DOP='0,0,0,0,0,1,0,6
.DEFI   .DEFLTI='0,0,0,0,0,1,0,4
.DEFI   .CREATI='0,0,0,0,0,1,0,5
.DEFI   .DTTAB='0,0,0,0,0,1,0,8
.DEFI   .DTRANS='0,0,0,0,0,1,0,9
.DEFI   .LOC='0,0,0,0,0,1,0,10
.DEFI   .DEFC='0,0,0,0,0,1,0,11
.DEFI   .DEXCLC='0,0,0,0,0,1,0,12
.DEFI   .DIBASE='0,0,0,0,0,1,0,13
.DEFI   .DOBASE='0,0,0,0,0,1,0,14
.DEFI   BMPY='46460103,3,0,0,0,0,0,1
.DEFI   LLSM='000001,0,0,0,0,0,0,1
.DEFI   FFB2='000000100004,0,0,0,0,0,0,1
.DEFI   CAB='3000004,1,0,0,0,0,0,1
.DEFI   CMO='300010002,1,0,0,0,0,0,1
.DEFI   BMPY1='46000102,1,0,0,0,0,0,1
.DEFI   BMPY2='00460101,2,0,0,0,0,0,1
.DEFI   BDIV='30000106,1,0,0,0,0,0,1
.DEFI   BFIX='7651000500474,'23,0,0,0,0,0,1
.DEFI   BFLO='00000007,0,0,0,0,0,0,1
.DEFI   MOVI='00006,'40,3,1,0,0,0,2
.DEFI   WR='00002,'40,3,1,0,0,0,2
.DEFI   WRS='00002000000004,'40,0,0,0,0,0,1
.DEFI   ARSB1='0000000004,'20,6,0,0,0,0,11
.DEFI   ARSB2='0000000004,'20,7,0,0,0,0,12
.DEFI   LRSB1='00000000044,'20,6,0,0,0,0,11
.DEFI   LRSB2='000000000044,'20,7,0,0,0,0,12
.DEFI   LEARSB1='00000000042,'20,6,0,0,0,0,11
.DEFI   LEARSB2='000000000024,'20,7,0,0,0,0,12
.DEFI   LAB='5100002,1,0,0,0,0,1,3
.DEFI   SAB='7500002,1,0,0,0,0,1,3
.DEFI   MOVA1='75,1,0,0,0,0,1,3
.DEFI   MOVA2='0075,2,0,0,0,0,1,4
.DEFI   MOVB1='51,1,0,0,0,0,1,3
.DEFI   MOVB2='0051,2,0,0,0,0,1,4
.DEFI   COMA1='01,1,0,0,0,0,1,3
.DEFI   COMA2='0001,2,0,0,0,0,1,4
.DEFI   COMB1='25,1,0,0,0,0,1,3
.DEFI   COMB2='0025,2,0,0,0,0,1,4
.DEFI   OR1='71,1,0,0,0,0,1,3
.DEFI   OR2='0071,2,0,0,0,0,1,4
.DEFI   AND1='55,1,0,0,0,0,1,3
.DEFI   AND2='0055,2,0,0,0,0,1,4
.DEFI   XOR1='31,1,0,0,0,0,1,3
.DEFI   XOR2='0031,2,0,0,0,0,1,4
.DEFI   ADD1='46,1,0,0,0,0,1,3
.DEFI   ADD2='0046,2,0,0,0,0,1,4
.DEFI   SUB1='30,1,0,0,0,0,1,3
.DEFI   SUB2='0030,2,0,0,0,0,1,4
.DEFI   LLSA1='62,1,0,0,0,0,1,3
.DEFI   LLSA2='0062,2,0,0,0,0,1,4
.DEFI   ZRO1='14,1,0,0,0,0,1,3
```

APPENDIX H. Instruction Table File for PMP-I (Continued)

FILE: PMPINST FORTRAN A 2/08/77 14:29 M.I.T. LINCOLN LABORATORY

```
.DEFI   ZRO2='0014,2,0,0,0,0,1,4
.DEFI   ONES1='16,1,0,0,0,0,1,3
.DEFI   ONES2='0016,2,0,0,0,0,1,4
.DEFI   ABSB='00460004,2,0,0,0,0,1,4
.DEFI   READ='0,'50,3,1,0,0,0,6
.DEFI   MOVM='00001,'10,0,0,0,0,0,10
.DEFI   ADD='0,4,1,0,0,0,0,9
.DEFI   JMPX='0000000000000010,4,0,0,0,0,0,1
.DEFI   JMKX='0000000000001010,4,0,0,0,0,0,1
.DEFI   SUB='0000000000000020,4,1,0,0,0,0,9
.DEFI   MOVSY='0000000000000030,4,0,0,0,0,0,1
.DEFI   LLSY4='0000000000000040,4,0,0,0,0,0,1
.DEFI   AND='0000000000000040,4,1,0,0,0,0,9
.DEFI   DREADY='0000000000000126,4,0,0,0,0,0,1
.DEFI   DREADXY='0000000000000127,4,0,0,0,0,0,1
.DEFI   OR='0000000000000060,4,1,0,0,0,0,9
.DEFI   NXTREAD='0000000000000131,4,0,0,0,0,0,1
.DEFI   ENABL='0000000000000100,4,0,0,0,0,0,1
.DEFI   XOR='0000000000000100,4,1,0,0,0,0,9
.DEFI   DSABL='0000000000000110,4,0,0,0,0,0,1
.DEFI   PSAVS='0000000000000120,4,0,0,0,0,0,1
.DEFI   LDX='0000000000000130,4,1,1,0,0,0,2
.DEFI   READS='000000000000004,'50,0,0,0,0,0,10
.DEFI   JMP='0000000000000020,4,2,1,1,0,1,2
.DEFI   MOVX='0000000000000140,4,0,0,0,0,0,5
.DEFI   JMK='0000000000001020,4,2,1,1,0,1,2
.DEFI   MOVY='0000000000000150,4,0,0,0,0,0,5
.DEFI   JMPXL='0000000000000137,4,2,1,1,0,1,2
.DEFI   INCX='0000000000000160,4,0,0,0,0,0,5
.DEFI   JMKXL='0000000000001137,4,2,1,1,0,1,2
.DEFI   DECX='0000000000000170,4,0,0,0,0,0,5
.DEFI   JMPXE='0000000000000136,4,2,1,1,0,1,2
.DEFI   ILSX='0000000000000200,4,0,0,0,0,0,5
.DEFI   COMP='0000000000000210,4,1,0,0,0,0,7
.DEFI   MEMX='0000000000000212,4,3,0,0,0,0,7
.DEFI   MEMY='0000000000000214,4,3,0,0,0,0,7
.DEFI   STAS='0000000000000216,4,3,0,0,0,0,7
.DEFI   LDS='0000000000000220,4,1,0,0,0,0,8
.DEFI   LDEI='0000000000000222,4,0,0,0,0,0,1
.DEFI   LDEIB='0000000000000223,4,0,0,0,0,0,1
.DEFI   LDEO='0000000000000224,4,1,1,0,0,0,2
.DEFI   JMKXE='0000000000001136,4,2,1,1,0,1,2
.DEFI   JMPXG='0000000000000140,4,2,1,1,0,1,2
.DEFI   JMKXG='0000000000001140,4,2,1,1,0,1,2
.DEFI   JMPAL='0000000000000133,4,2,1,1,0,1,2
.DEFI   JMKAL='0000000000001133,4,2,1,1,0,1,2
.DEFI   JMPAE='0000000000000132,4,2,1,1,0,1,2
.DEFI   JMKAE='0000000000001132,4,2,1,1,0,1,2
.DEFI   JMPAG='0000000000000134,4,2,1,1,0,1,2
.DEFI   JMKAG='0000000000001134,4,2,1,1,0,1,2
.DEFI   JMPTX='0000000000000135,4,2,1,1,0,1,2
.DEFI   JMKTX='0000000000001135,4,2,1,1,0,1,2
.DEFI   JMPOVF='0000000000000160,4,2,1,1,0,1,2
.DEFI   JMKOVF='0000000000001160,4,2,1,1,0,1,2
.DEFI   DREADX='0000000000000125,4,1,1,0,0,0,2
```

APPENDIX H. Instruction Table File for PMP-I (Continued)

FILE: PMPINST FORTRAN A 2/08/77 14:29 M.I.T. LINCOLN LABORATORY

```
.DEFI   CLRIO='0000000000000236,4,4,1,0,0,0,2
.DEFI   SETIO='0000000000000231,4,4,1,0,0,0,2
.DEFI   SETMUX='0000000000000234,4,4,0,0,0,0,7
.DEFI   MSKIO='0000000000000232,4,4,0,0,0,0,7
.DEFI   HALT='0000000000000230,4,0,0,0,0,0,1
.DEFI   SETAUX='0000000000000237,4,0,0,0,0,0,1
.DEFI   CLRAUX='0000000000000240,4,0,0,0,0,0,1
.DEFI   JMKDEI='0000000000001122,4,2,1,1,0,1,2
.DEFI   JMKDEO='0000000000001123,4,2,1,1,0,1,2
.DEFI   JMKDHP='0000000000001124,4,2,1,1,0,1,2
.DEFI   JMPDEI='0000000000000122,4,2,1,1,0,1,2
.DEFI   JMPDEO='0000000000000123,4,2,1,1,0,1,2
.DEFI   JMPDHP='0000000000000124,4,2,1,1,0,1,2
.DEFI   LDPM='0000000000000050,4,5,1,0,0,0,2
.DEFI   PINTS='0000000000000121,4,0,0,0,0,0,1
.DEFI   JMPXNE='0000000000000225,4,2,1,1,0,1,2
.DEFI   JMKXNE='0000000000001225,4,2,1,1,0,1,2
.DEFI   JMPXLE='0000000000000226,4,2,1,1,0,1,2
.DEFI   JMKXLE='0000000000001226,4,2,1,1,0,1,2
.DEFI   JMPXGE='0000000000000227,4,2,1,1,0,1,2
.DEFI   JMKXGE='0000000000001227,4,2,1,1,0,1,2
.DEFI   LLS4='0000000000000060,4,1,1,0,0,0,2
.DEFI   SWAPSY='0000000000000070,4,0,0,0,0,0,1
.DEFI   MOVSYXS='0000000000000150,4,0,0,0,0,0,1
.DEFI   LEASX='0000000000000240,4,0,0,0,0,0,5
.DEFI   MEMXSY='0000000000000250,4,3,0,0,0,0,7
.DEFI   MEMXMSY='0000000000000252,4,3,0,0,0,0,7
.DEFI   MEMXS='0000000000000254,4,3,0,0,0,0,7
.DEFI   INCXMX='0000000000000256,4,3,0,0,0,0,7
.DEFI   MOVXSTS='0000000000000260,4,3,0,0,0,0,7
.DEFI   INCXSTS='0000000000000262,4,3,0,0,0,0,7
.CREATI
```

APPENDIX I. Sample Program Listing

```
LINE  ADDR    OBJECT                SOURCE

   1                                /*************************************************************************
   2                                /
   3                                /     THE FOLLOWING SAMPLE PROGRAM LISTING IS MERELY A SMALL PORTION
   4                                /     OF A MUCH LARGER PROGRAM.  THIS PORTION OF CODE COMPUTES THE
   5                                /     MAGNITUDES OF 16 COMPLEX FAST FOURIER TRANSFORM (FFT) OUTPUTS, AND
   6                                /     THEN COMPUTES THE LOGARITHMS OF THESE RESULTS.
   7                                /
   8                                /     NOTICE THE PROGRAMMING CONVENTION TO REFERENCE PE RAM LOACATION
   9                                /     'N' BY 'PMN' RATHER THAN BY MERELY USING 'N' (WHERE 'PMN' IS
  10                                /     DEFINED WITH A .DEFC).  THIS ALLOWS ALL REFERENCES TO A
  11                                /     RAM LOCATION TO APPEAR IN THE CROSS REFERENCE TABLE.  A SIMILAR
  12                                /     CONVENTION WAS CHOSEN TO REFERENCE CONTROLLER XMEM LOCATION 'N'
  13                                /     AS 'XMN'.
  14                                /
  15                                /*************************************************************************
  16
  17
  18
  19
  20
  21                                      .DEFC   PM60='60
  22                                      .DEFC   PM61='61
  23                                      .DEFC   PM344='344
  24                                      .DEFC   PM345='345
  25                                      .DEFC   PM364='364
  26                                      .DEFC   PM365='365
  27
  28
  29                                /*************************************************************************
  30                                /
  31                                /     CALCULATE THE MAGNITUDE OF EACH COMPLEX SAMPLE STORED IN RAM LOCAT
  32                                /     0-'37 BY APPROXIMATION.  THE ALGORITHM USED IS AS FOLLOWS:
  33                                /     NOTE:  ML REFERS TO LARGEST ABS. VAL. OF REAL & IMAG PART, MS TO
  34                                /        SMALLEST.
  35                                /
  36                                /     1.  CALCULATE ABSOLUTE VALUE OF BOTH REAL & IMAGINARY PARTS
  37                                /     2.  SET MAG=LARGER(ML,(ML-ML/8-MS/2))
  38                                /
  39                                /     THE OUTPUT MAGNITUDES WILL BE STORED IN THE EVEN-ADDRESSED
  40                                /     LOCATIONS (I.E. WHERE THE I-VALUES WERE INITIALLY STORED).
  41                                /
  42                                /*************************************************************************
  43
  44
  45 00000  0000000000001300000     MAG_BEGIN:    LDX  0                  /INIT POINTERS TO LOCATION 0
  46 00001  0000000000001420000           MOVX    S
  47
  48 00002  0014000000142562000     MAG_LOOP:   ZPOI   A2;READS  B2;INCX  S     /B2=I-VALUE;A2=0;X,S=X,S+1
  49 00003  0046060400140542000           ABSB    M;READS  B2;MOVX    S   /M=ABS(I-VALUE);B2=Q-VALUE
  50 00004  0046160414000164000           MOVM    B1;ABSB   M;INCX   X       /B1=ABS(I-VALUE);M=ABS(Q-VALUE)
```

APPENDIX I. Sample Program Listing (Continued)

```
LINE  ADDR        OBJECT                                SOURCE

  51  00005   5100100014004000000     MOVB1  A1;MOVM   B1            /A1=ABS(I-VALUE);B1=ABS(Q-VAL
  52  00006   3000004000000000000     CAB
  53  00007   5100022000000000000     LAB    M                      /M=LARGEST
  54  00010   7500122014140000000     MOVM B1,B2;SAB     M         /M=SMALLEST;B1,B2=LARGEST
  55
  56                                /*****   COMPUTE M1-M1/8+MS/2 IN ALU2   *****
  57  00011   5151000000466000000     MOVB1  A1;MOVB2  A2;LRSB2   3     /A1,A2=M1;B2=M1/8
  58  00012   0030100000142000000     SUB2   A2;MOVM   B2                /A2=M1-M1/8;B2=MS
  59  00013   0000000000470000000     LRSB2  1                           /B2=MS/2
  60  00014   0046060000000000000     ADD2   M                           /M=M1-M1/8+MS/2
  61  00015   0000100014000210003?     MOVM   B1;COMP   '37               /B1=M1-M1/8+MS/2;ALL 16 DONE?
  62  00016   3000004000000137000?     CAB      ;JMPXI  MAG_LOOP         /NOT YET--CONTINUE
  63  00017   5100022000000000000     LAB      M                         /M=LARGER(M1,(M1-M1/8+MS/2))
  64                                                          /FINAL MAG IN M
  65  00020   0000200000000566000?     WRS;INCX   X,S          /STORE RESULT
  66
  67                                /*****************************************************************
  68                                /
  69                                /     ROUTINE TO COMPUTE 20LOG10 OF MAGNITUDED RESULTS.
  70                                /
  71                                /     THE 16 MAGNITUDED NUMBERS ARE EXPECTED IN RAM LOCATIONS 0-'37,
  72                                /     IN THE EVEN-NUMBERED LOCATIONS, AND
  73                                /     OUTPUTS ARE PLACED IN RAM LOCS ('345-'364) BY DIVIDING THE ADDRESS
  74                                /     OF THE INPUT BY 2, AND ADDING '345.  THIS PUTS OUTPUTS IN THE SAME
  75                                /     ORDER AS INPUTS; I.E. CELLS 9-16, FOLLOWED BY CELL 1-8.  WHEN ALL
  76                                /     LOGS HAVE BEEN TAKEN, CELL 8 IS ALSO PUT IN RAM('344) AND CELL 9
  77                                /     IS ALSO PUT IN RAM('365).  THIS AVOIDS WRAP-AROUND DIFFICULTIES
  78                                /     DURING THE INTERPOLATION ALGORITHM.
  79                                /
  80                                /     ALGORITHM USED FOR TAKING LOGARITHM IS:
  81                                /     L-SHIFT THE DATA UNTIL THE MSB=1, COUNTING HOW MANY SHIFTS WERE
  82                                /     DONE (BFLO DOES THIS NICELY).  THIS GIVES THE BIT # OF THE FIRST
  83                                /     NON-ZERO BIT.  THIS BIT # IS MPY'ED BY 16, AND THEN THE NEXT 4
  84                                /     DATA BITS ARE SUBTRACTED.  THIS RESULT, THEN, IS SUBTRACTED
  85                                /     FROM 256.  THE RESULT IS A NUMBER SCALED FROM 0 (-90DB) TO '377
  86                                /     (0DB).  IN SUMMARY, OUTPUT=256-((16*LEFT-MOST MAG BIT#)-(NEXT 4
  87                                /     BITS))
  88                                /
  89                                /
  90                                /     RAM LOCATIONS USED:
  91                                /     (0-'37) INPUT VALUES (MAGNITUDES) IN THE EVEN-ADDRESSED LOCATIONS
  92                                /     ('345-'364) OUTPUT VALUES (20LOG10(MAGNITUDES)) STORED
  93                                /     ('60)  +1 STORED HERE
  94                                /     ('61)  +6 STORED HERE FOR SCALING(MAKE MAX DC VALUE =0DB)
  95                                /     ('344) STORE VALUE FROM DOPPLER CELL 8 HERE ALSO(FOR WRAP-AROUND)
  96                                /     ('365) STORE VALUE FROM DOPPLER CELL HERE ALSO (FOR WRAP-AROUND
  97                                /
  98
  99                                /*****************************************************************
 100
 101  00021   0000000000001300000     LOG_BEGIN:    LDX  0    /INIT BUFFER PTR TO BEGIN OF INPUT BUFFER
 102  00022   0000000000001420000     MOVX  S
```

APPENDIX I. Sample Program Listing (Continued)

```
LINE  ADDR    OBJECT                          SOURCE

  103
  104                                 /*****   LOAD APPROPRIATE INPUT DATA   *****
  105 00022   0000000014000E440000    LOG_LOOP:   READS   B1;LEASX  X  /B1=INPUT DATA
  106
  107                                 /***   ADDR FOR OUTPUT CALC'ED BY ADDING '340 TO R-SHIFTED 7 BIT
  108                                 /   (LEASX 23 BITS)   ****************************************
  109                                 /*****   DETERMINE BIT # OF 1RST NON-ZERO BIT   *****
  110 00024   5114020014142244006O        MOVB1  M;ZPO2     A2;READ    RM60,B2,B1;LEASX  X   /A2=0;B1,B2=+1;M=INP
  111 00025   5100000700004244000O            BFLO;MOVB1   A1;LEASX  X      /GENERATE 256('400) IN A1
  112 00026   6200000700004244000O            BFLO;LLSA1   A1;LEASX  X
  113 00027   6200000700004244000O            BFLO;LLSA1   A1;LEASX  X
  114 00030   6200000700004244000O            BFLO;LLSA1   A1;LEASX  X
  115 00031   6200000700004244000O            BFLO;LLSA1   A1;LEASX  X
  116 00032   6200000700004244000O            BFLO;LLSA1   A1;LEASX  X
  117 00033   6200000700004244000O            BFLO;LLSA1   A1;LEASX  X
  118 00034   6200000700004244000O            BFLO;LLSA1   A1;LEASX  X
  119 00035   6200000700004244000O            BFLO;LLSA1   A1;LEASX  X
  120 00036   0000000714000244006I            BFLO;READ    RM61,B1;LEASX  X      /SUBTRACT 6 TO SCALE MAX DC=0DB
  121 00037   3000000700004244000O            BFLO;SUB1    A1;LEASX  X
  122 00040   0000000700000244000O            BFLO;LEASX  X
  123 00041   0000000700000244000O            BFLO;LEASX  X
  124 00042   0000000700000244000O            BFLO;LEASX  X
  125 00043   0000000700000244000O            BFLO;LEASX  X
  126 00044   0000000700000244000O            BFLO;LEASX  X
  127
  128                                 /***** A2 NOW CONTAINS BIT #;MPY BY 16(L-SHIFT 4) AND GET NEXT
  129                                 /        4 BITS IN B2              *****
  130 00045   0062010000002244000O        LLSA2   A2;LLSM;LEASX  X   /OMIT MSB OF M (FIRST '1')
  131 00046   0062100000142244000O        LLSA2   A2;MOVM   B2;LEASX  X
  132 00047   0062000000442244000O        LLSA2   A2;LRSB2  4;LEASX  X  /PUT B2=NEXT 4 BITS IN LSB POSITIONS
  133 00050   0062000000442244000O        LLSA2   A2;LRSB2 4;LEASX  X
  134 00051   0000000000442244000O        LRSB2   4;LEASX  X
  135 00052   0000000000440002034S        LRSB2   4;ADD  '345,S
  136 00053   0000000000442244000O        LRSB2   4;LEASX  X
  137 00054   0030060000000000000O        SUB2    M    /M=(16*(BIT#)-(NEXT 4 BITS))
  138 00055   0014160014000000000O        MOVM    B1;ZPO2    M
  139 00056   3000100014004210003I        SUB1    A1;MOVM   B1;COMP    '37    /M=256-(16*(BIT#)-(NEXT 4 BITS
  140 00057   3000004000000137002I        CAB     ;JMPX1   LOG_LOOP     /CONTINUE LOOPING IF NOT YET DONE
  141 00060   5100022000000000000O        LAB     M                /SET SO SCALING DOESN'T PRODUCE A NEG VALU
  142 00061   0000200000000406000I        WRS;ADD     2,X,S  /STORE DATA IN OUTPUT BUFFER
  143 00062   0000000014000000345I        READ  RM345,B1    /***SET UP ENDS FOR WRAP-AROUND*****
  144 00063   5100020014000000364I        MOVB1  M;READ  RM364,B1
  145 00064   5100220000000000365I        WR  RM365;MOVB1  M
  146 00065   0000200000000000344I        WR  RM344
```

APPENDIX I. Sample Program Listing (Continued)

CROSS REFERENCE TABLE

NAME	ADDRESS	DEFINED	REFERENCES
LOG_BEG	00021	101	
LOG_LOO	00023	105	140
MAG_BEG	00000	45	
MAG_LOO	00002	48	62
RM344	00344	23	146
RM345	00345	24	143
RM364	00364	25	144
RM365	00365	26	145
RM60	00060	21	110
RM61	00061	22	120

0 STATEMENTS FLAGGED IN THIS ASSEMBLY

APPENDIX J Error Messages Produced at Assembly Time

Following is a description of all error messages produced by the cross-assembler, with possible corrective actions suggested. All error messages are output immediately before the source line producing the error.

1. *****EXCLUSIVITY SET CONFLICT

   Two or more mutually exclusive instructions have been coded for the same program memory word. The instructions have been inclusive OR'ed which probably results in an instruction other than either of the two conflicting instructions.

2. *****MORE THAN 1 IC USED

   Two instructions intended for the same program memory word require the use of the large IC field. The two IC's have been inclusive OR'ed, probably producing a third IC.

3. *****ILLEGAL COMBINATION

   Two instructions are trying to set the same bit, typically indicating multiple use of registers, wires, etc. The instructions will still be OR'ed, with proper execution doubtful.

4. *****SOURCE PROGRAM TOO LONG

   An attempt was made to assemble code for a memory location whose address is greater than the upper limit specified by the .DARCH pseudo-op. Assembly is terminated.

5. *****UNDEFINED LABEL name

   A forward reference was made to an IC called name, but at the end of assembly the value of name had not yet been defined.

6. *****UNKNOWN OPCODE opcode

The instruction mnemonic opcode was coded, but it does not appear in the table of defined instruction mnemonics.

7. *****INVALID OPERAND operand for opcode

The instruction mnemonic opcode appeared with operand coded as an operand, but operand is not an element of the operand set for this instruction. operand is ignored.

8. *****INSTRUCTION opcode ALREADY DEFINED

A .DEFI attempted to define the instruction mnemonic opcode, but opcode has already been defined. The new definition will be ignored.

9. ****INSTRUCTION TABLE FULL

An attempt to define an instruction mnemonic with a .DEFI was made, but there was no room in the instruction table for this mnemonic. The .DEFI will be ignored.

10. *****INVALID CANONIC CLASS FOR opcode

A .DEFI pseudo-op attempted to define the instruction menmonic opcode, but specified that opcode belonged to a nonexistent canonic class (where number of canonic classes was specified in the .DARCH pseudo-op).

11. *****INVALID ATTEMPT TO CHANGE ADDRESS

The operand of the .LOC pseudo-op is invalid. Usually this indicates either an attempt to set the address counter to a value less

than the current value or to a value greater than the maximum program memory location as specified by the .DARCH pseudo-op.

12. *****CONSTANT operand OUT OF LIMITS

The immediate constant operand is outside the limits for the current instruction.

13. *****MISSING OPERAND FOR opcode

The instruction opcode has been coded with an insufficient number of operands. Depending on the instruction mnemonic the cross-assembler may or may not examine operands that were coded.

14. *****TOO MANY OPERANDS FOR opcode

The instruction mnemonic opcode has more than the maximum allowable number of operands. Only the first n operands will be examined, where n is the maximum number of allowable operands.

15. *****INVALID EXCLUSIVITY SET FOR opcode

An attempt to define opcode with a .DEFI pseudo-op contains an illegal exclusivity set value. The definition of this instruction mnemonic is ignored.

16. *****REQUIRES operand1 OR operand2

Either operand1 or operand2 is required for this instruction mnemonic but was not coded. The available operands are assembled.

17. *****WRONG PLACE FOR OPERAND operand

operand is a valid operand for the current instruction and its position in the list of operands is essential to its meaning. The instruction appears here with operand coded but not in the proper position. The cross-assembler will ignore operand.

18. *****INVALID OCTAL STRING FOR name

name is either an opcode being defined with a .DEFI pseudo-op or an operand being defined with a .DOP pseudo-op. The actual bit string value to be assigned to name must be coded as an octal constant beginning with an apostrophe ('), but is coded illegally here. The .DEFI or .DOP is ignored.

19. *****RESERVED MNEMONIC name USED AS CONSTANT

name is a reserved mnemonic but the user has attempted to use it either as a label or as an operand which must be an immediate constant. If it is being defined as a label the definition is ignored and if name appears as an operand a value of zero is used.

20. *****name MULTIPLY DEFINED

Either name is a label being defined which already has been defined or a reference is being made to a label for which multiple definitions have been given. If a definition is being given it will be ignored, and thus a reference to a multiply defined label will retrieve the value first assigned to that label.

21. *****SYMBOL TABLE OVERFLOW FOR name

An attempt to define the label name was unsuccessful since there was no longer any room available in the symbol table. The user will need to omit some labels. Currently the number of symbols allowed is 1023 minus the number of reserved mnemonics.

22. *****UNKNOWN CONSTANT name

A reference to name appeared but name has not yet been defined and the current instruction does not allow forward referencing. A value of zero will be used.

23. *****INVALID CONSTANT name

The immediate constant name was coded but cannot be deciphered. Typically this indicates a numeric constant containing a non-numeric character or a number not allowed in the current base.

24. *****CROSS-REFERENCE TABLE FULL

A label was referenced but there was no room available to save this in the cross-reference table. The program memory word is assembled correctly, but the cross-reference listing will not reflect this variable reference.

25. *****IC LIMITS EXCEEDED BY CONSTANT WITH FORWARD REFERENCES AT ADDRESSES numbers

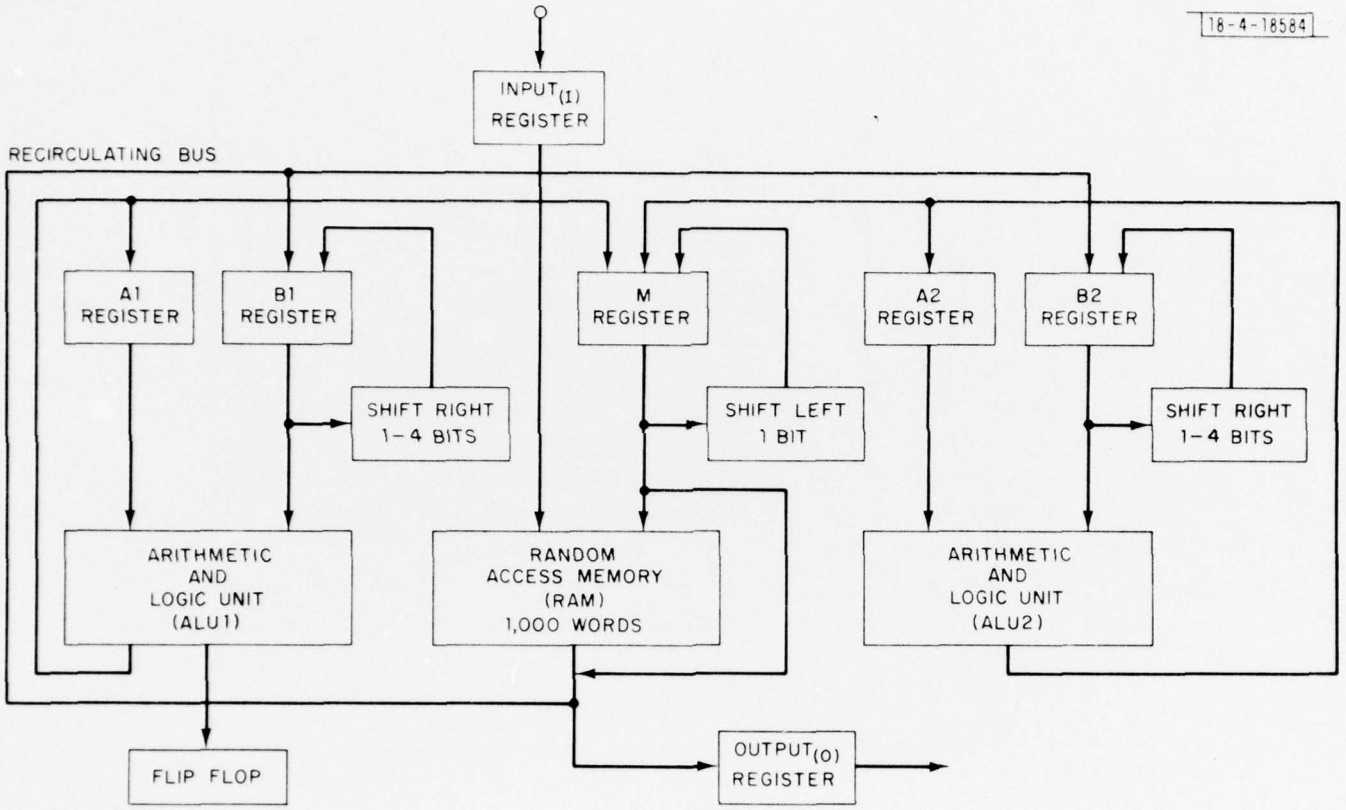
This message is produced when a label which has been forward referenced is defined but is not within allowable limits. numbers is a list of all addresses where this label has been referenced. The address fields of all addresses listed will not be updated.

APPENDIX K. ALU Function Select Table for PMP-I

Selection S_3 S_2 S_1 S_0	ACTIVE-HIGH DATA: M-H Logic Functions	ACTIVE-HIGH DATA: M-L; ARITHMETIC OPERATIONS: C_n = 1 = H (no carry)	ACTIVE-HIGH DATA: M-L; ARITHMETIC OPERATIONS: C_n = 0 = L (with carry)
L L L L	F=$\overline{A}$	F=A	F=A PLUS 1
L L L H	F=A+$\overline{B}$	F=A+$\overline{B}$	F=(A+B) PLUS 1
L L H L	F=$\overline{A}$B	F=A+B	F=(A+$\overline{B}$) PLUS 1
L L H H	F=0	F=MINUS 1 (2's COMPL)	F=ZERO
L H L L	F=$\overline{AB}$	F=A PLUS A$\overline{B}$	F=A PLUS A$\overline{B}$ PLUS 1
L H L H	F=$\overline{B}$	F=(A+B) PLUS A$\overline{B}$	F=(A+B) PLUS A$\overline{B}$ PLUS 1
L H H L	F=A $\oplus$ B	F=A MINUS B MINUS 1	F=A MINUS B
L H H H	F=A$\overline{B}$	F=A$\overline{B}$ MINUS 1	F=A$\overline{B}$
H L L L	F=$\overline{A}$+B	F=A PLUS AB	F=A PLUS AB PLUS 1
H L L H	F=$\overline{A}$ $\oplus$ $\overline{B}$	F=A PLUS B	F=A PLUS B PLUS 1
H L H L	F=B	F=(A+$\overline{B}$) PLUS AB	F=(A+$\overline{B}$) PLUS AB PLUS 1
H L H H	F=AB	F=AB MINUS 1	F=AB
H H L L	F=1	F=A PLUS A*	F=A PLUS A PLUS 1
H H L H	F=A+$\overline{B}$	F=(A+B) PLUS A	F=(A+B) PLUS A PLUS 1
H H H L	F=A+B	F=(A+$\overline{B}$) PLUS A	F=(A+$\overline{B}$) PLUS A PLUS 1
H H H H	F=A	F=A MINUS 1	F=A

* Each bit is shifted to the next more significant position.

APPENDIX L. PMP-I Processor Element (PE) Architecture



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: ESD-TR-77-72	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: A General-Purpose Cross-Assembler for Producing Absolute Binary Object Code	5. TYPE OF REPORT & PERIOD COVERED: Technical Note
6. PERFORMING ORG. REPORT NUMBER: Technical Note 1977-20	
7. AUTHOR(S): Paul R. Kretz	8. CONTRACT OR GRANT NUMBER(S): F19628-76-C-0002
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Lincoln Laboratory, M.I.T. P.O. Box 73 Lexington, MA 02173	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: Program Element No. 63208F Project No. 2027
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Systems Command, USAF Andrews AFB Washington, DC 20331	12. REPORT DATE: 6 April 1977 13. NUMBER OF PAGES: 76
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Electronic Systems Division Hanscom AFB Bedford, MA 01731	15. SECURITY CLASS. *(of this report)*: Unclassified 15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

TN-1977-20

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

cross-assembler
binary object code
microcode
PMP-I
pseudo-op definitions
metalanguage commands
Conversational Monitor System

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

A general-purpose cross-assembler is described. The cross-assembler, written in PL/1, has been implemented on an IBM 370/168 using the time-sharing Conversational Monitor System (CMS). Absolute binary object code will be produced. Although the cross-assembler has been designed with the intention of assembling code for various microprogrammable machines, even code for conventional minicomputers has been assembled.

Use of the cross-assembler is discussed assuming a CMS environment. Included are the disk-resident files to facilitate an assembly. Various pseudo-ops, or assembler control statements, are used to describe the machine for which an assembly is done. An example of using the cross-assembler for a parallel microprogrammable digital signal processor (PMP-I) is discussed.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

207650